

37

LE CANARD QUI A DU CARACTERE

JUIN 1987

EDITORIAL

Ecrire à l'aide d'un traitement de texte est plus valorisant et plus commode qu'avec une simple machine à écrire. Pourtant il se vend toujours des machines à écrire, parfois plus chères qu'un ordinateur. Petit pastiche publicitaire: "Mr, échangeriez-vous votre traitement de texte contre deux machines à écrire ECRIOBROUILLON?". Non! Echangeriez-vous JEDI contre deux MIEUXDISANTUNCULTUREL? ...

Et si nous parlons informatique dans JEDI, nous savons également exploiter cette informatique pour mieux en parler. Dans un univers où PAO, DAO, EAO, TELEMATIQUE, RESEAUX, IA, SYSTEMES EXPERTS, etc... seront de plus en plus sur les cartes de visite d'une majorité d'entre vous, la place d'une revue comme JEDI est de plus en plus nécessaire. Car chacun aura toujours une petite question à résoudre; car le meilleur mode d'emploi laissera toujours dans l'ombre une instruction obscure; car le LOGICIEL devient plus important que le matériel.

Alors encore une fois, et pour que ce ne soit pas la dernière, si vous avez des idées, faites-les connaître, publiez-les dans JEDI.

SOMMAIRE

INFORMATION

La revue JEDI a un besoin urgent de transfusion de matière rédactionnelle pour rattraper son retard de publication. Cette transfusion doit s'opérer sans prescription médicale. En cas d'échec, votre dealer ne répond plus de la fourniture régulière de votre dose de nouveaux langages.

Cette transfusion peut s'opérer sous forme de lettres, listings, ou mieux, sous forme de fichiers ASCII, WORSTAR, WORDPERFECT.



Toute reproduction, adaptation, traduction partielle du contenu de ce magazine sous toutes les formes est vivement encouragée, à l'exception de toute reproduction à des fins commerciales. Dans le cas de reproduction par photocopie, il est demandé de ne pas masquer les références inscrites en bas de page, et dans les autres cas de citer l'ASSOCIATION JEDI (association loi 1901).

Nos coordonnées: ASSOCIATION JEDI 17, rue de la Lancette 75012 PARIS
tel président: (1) 43.40.96.53
tel secrétaire: (1) 46.56.33.67

FORTHLOG II
par Marc PETREMANN

Au vu des commandes consecutives à la lettre du secretaire du mois d'avril 87, il est de notre devoir de vous expliquer en détail les fonctionnalités de FORTHLOG II. Cet article n'est pas un cours d'initiation aux systemes experts, mais seulement un commentaire qui survole les possibilités de ce nouveau logiciel.

LE PRODUIT FORTHLOG II

Dans le n°23 du magazine, nous diffusions un générateur de systeme expert nommé FORTHLOG. Conçu au départ pour tourner sur un système COMMODORE, il est adaptable sur les autres versions de FORTH. C'est ce que n'a pas manque de faire l'auteur, Francis LEY, qui l'a installe en F83 sous MSDOS et en y apportant des améliorations.

Parrallèlement, le manuel a été tapé, illustré et indexé. Il sera mis à jour en fonction des diverses modifications suggérées par les utilisateurs, le groupe de travail ou le concepteur.

Actuellement, des sociétés renommées utilisent déjà FORTHLOG II, dont CAP SOGETI SYSTEMES (division Mulhouse), UNI-META (Longwy) et le MINISTERE DES FINANCES (Service des Bâtiments, Paris).

Pour rappel, FORTHLOG II travaille en chainage avant et arriere, utilise des coefficients de pondération, la notation algébrique infixée et le chainage des bases de connaissances.

Le produit FORTHLOG II comprend:

- un programme F83 sans les extensions developpeur (META86, KERNEL86, CPU8086, EXTEND86, UTILITY).
- le logiciel FORTHLOG II (blocs 1 à 100), des exemples d'utilisation (blocs 120 à 200).

PRISE EN MAIN

La maitrise de FORTHLOG II nécessite déjà une certaine connaisance de systèmes experts. Pour ce faire, reportez-vous aux ouvrages existants dans ce domaine.

FORTHLOG II est encore entaché de petits inconvénients qui disparaitrons dans les versions futures. Il faut notamment compiler le bloc de liaison d'une base de connaissance avant de l'exploiter. Ce bloc de liaison contient un certain nombre de paramétres qu'il faut initialiser manuellement. Exemple de bloc de liaison de la base de connaissance du calcul d'un bulletin de paie:

```
Scr # 120
\ E9 PARAMETRES FORTHLOG          ( LEY)
( NUMERO DU 1ER BLOC POUR GROUPE DE REGLE DE NIVEAU 1,2,3 )
   0   #PRE_NIV1 !     122 #PRE_NIV2 !     124 #PRE_NIV3 !

( NOMBRE MAXIMUM DE REGLE POUR GROUPE DE NIVEAU 1,2,3 )

   0   #NB_REG1 !      12  #NB_REG2 !      12  #NB_REG3 !

( NUMERO DU 1ER BLOC POUR BASE DE FAITS )
   121 #PRE_FAIT !

( NOMBRE MAXIMUM DE BLOC POUR BASE DE FAITS )
   1    #NB_BLOC_F !
```

Pour activer la base de connaissance du calcul d'un bulletin de paie, il faut taper **120 LOAD**.

Il existe trois niveaux de régles:

- les regles UNIVERS
- les regles CONTEXTE; dans notre exemple, elles se situent dans les blocs 122 et 123.
  - les régles RESULTAT; dans notre exemple elles occupent les blocs 124 et 125.

Attention, cet exemple est défini deux fois dans FORTHLOG. Une version en notation algébrique infixée et une en notation RPN (Notation polonaise inverse), ceci pour permettre une comparaison.

La base de faits est définie dans le bloc 121. Les faits initialisent les variables utilisees dans les régles, mais permettent également de contrôler les actions du systeme expert. Exemple:

```
  $AFF Quel type de collaborateur ? " CR
  $AFF Réponses possibles -TECHNICIEN , CADRE- "
                                          CR PERSON $?AF CR
  $AFF Salaire de base en Frs ? " CR SAL_BASE $?AFN CR
  $AFF Nbre de jour de congé ? " CR CONGE $?AFN CR
  $AFF Nbre de jour d'absence ? " CR ABSENCE $?AFN CR
  $AFF Nbre de ticket restaurant ? " CR T_RESTO $?AFN
  CR  17 TAUX_SOC $AF-N
  5 TAUX_RETR $AF-N
                          ;
```

Ici, la primitive FORTHLOG **$AFF** permet d'afficher le contenu d'une chaine de caractères en mode interprétation. Sans elle, tout mot rencontre ne faisant pas partie du vocabulaire courant serait transforme en une variable de type **$VARIABLE**, ce qui est le cas de **PERSON**, **SAL_BASE**, **CONGE**, etc...

Les autres mots FORTH sont exécutés de manière normale. La syntaxe des commandes FORTHLOG est volontairement "complexe", ceci pour éviter les confusion entre les noms de $variable, les primitives FORTH et vos propres définitions. Elles sont pour la majorité précédées du signe **$** donc facilement identifiables:

```
 $?AF      $?AFN     $"        $'        $<
 $>        $=        $AF       $AF-N     $AFF
 $ATTACH   $CF       $FIN      $NON      $OU
 $STOP
```

Echappent à cette syntaxe, les primitives:

```
  SI        ALORS
```

Dans les faits, rien n'empêche de generer un dessin si vous avez implanté les fonctions graphiques en F83. Il en est de même pour les interfaces série, //, souris, palette graphique, etc... Le résultat d'une action peut être affecté à une $variable. Exemple, soit le mot JOYSTICK chargé de délivrer la position du manche:

```
  : JOYSTICK  ( --- n  de [0..7]
     ...définition de JOYSTICK...  ;
```

Dans les faits, on utilisera JOYSTICK ainsi:

```
  JOYSTICK MANCHE $?AFN
```

et la $variable **MANCHE** utilisée dans une règle:

```
  SI MANCHE 1 $=
  ALORS $AFF Vous allez en direction NE " CR
```

On peut également, apres avoir declaré les variables adéquates dans les faits, decrire les règles:

```
  SI MANCHE 1 $=
  ALORS $[ SENS @ + 1 ]$
  SI MANCHE 7 $=
  ALORS $[ SENS @ - 1 ]$
```

où nous illustrons également l'emploi de la notation infixée.

Voici pour l'exemple du calcul d'un bulletin de paie, les règles contrôlant l'expertise:

```
SI SAL_BASE 0 $=
ALORS CR $AFF SALAIRE DE BASE ERRONE STOP !!" $STOP
SI CONGE 0 $>
ALORS $[ SAL_BASE / 21 ]$       VAL_JOUR_CONGE $AF-N
SI CONGE 0 $>
ALORS  $[   VAL_JOUR_CONGE  *  CONGE  ]$  RET_CONGE  $AF-N
                                    RET_CONGE PLUS_CONGE $AF-N

SI ABSENCE 0 $>
ALORS $[ SAL_BASE / 22 ]$       VAL_JOUR_ABS $AF-N
SI SAL_BASE
ALORS SAL_BASE BRUT $AF-N
SI ABSENCE 0 $>
ALORS $[ VAL_JOUR_ABS * ABSENCE ]$ RET_ABS $AF-N $[ BRUT -
                                        RET_ABS ]$ BRUT $AF-N

SI BRUT
ALORS $[ ( BRUT / 100 ) * TAUX_SOC ]$ CHARG_SOC $AF-N
```

```
SI BRUT
ALORS $[ ( BRUT / 100 ) * TAUX_RETR ]$ CHARG_RETR $AF-N
SI BRUT
ALORS $[ ( BRUT - CHARG_SOC ) - ( CHARG_RETR ) ]$ NET $AF-N
SI T_RESTO 0 $>  &  PERSON $' TECHNICIEN' $=
ALORS $[ T_RESTO * 11 ]$    VAL_RESTO $AF-N
SI T_RESTO 0 $>   &   PERSON $' CADRE' $=
ALORS $[ T_RESTO * 13 ]$    VAL_RESTO $AF-N
SI VAL_RESTO 0 $>
ALORS $[ NET - VAL_RESTO ]$  PAYER $AF-N
```

et qui sont les règles décrivant le contexte.

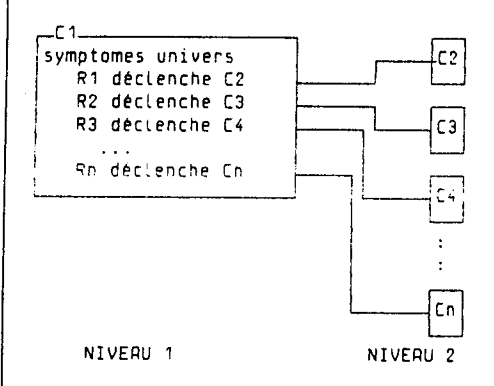
```
SI PAYER
ALORS $AFF APPOINTEMENT " SAL_BASE 5 .R CR
SI PAYER
ALORS $AFF CONGES RET.  " PLUS_CONGE 15 .R CR
SI PAYER
ALORS $AFF CONGES PAYES " PLUS_CONGE 5 .R CR
SI PAYER
ALORS $AFF ABSENCE RET. " RET_ABS 15 .R CR
SI PAYER
ALORS $AFF BRUT FISCAL  " BRUT 5 .R CR
SI PAYER
ALORS $AFF CHARGES SOC. " CHARG_SOC 15 .R CR
SI PAYER
ALORS $AFF RETRAITE     " CHARG_RETR 15 .R CR
SI PAYER
ALORS $AFF NET " NET 5 .R CR
SI PAYER
ALORS $AFF RESTRAURANT  " VAL_RESTO 15 .R CR
SI PAYER
ALORS $AFF NET A PAYER  " PAYER 5 .R CR
```

et qui sont les règles délivrant les résultats.

Cette base de connaissance ne peut être traitee en chainage arriere, car elle dépasse la simple logique propositionnelle (nommée également logique d'ordre 0).

En chainage avant, on vérifie toutes les règles à partir des faits existants et on poursuit l'expertise jusqu'à "saturation" des règles.

Dans le cas de FORTHLOG II, une règle de niveau 1, c'est à dire du niveau CONTEXTE peut déclencher l'attachement de une ou plusieurs bases de connaissances de type 2 et 3:

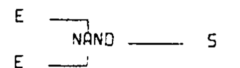


Le nombre de règles pouvant être ainsi déclenchées est "infini" (dépend de la capacité du disque, pres de 20Mo pour un disque dur...).

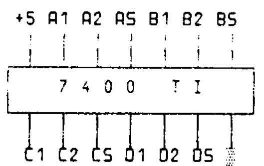
Les règles doivent faire partie du même fichier FORTH géré sous forme de blocs. La limitation du nombre de règles (actuellement 100) peut être modifié en retouchant les variables du programme source FORTHLOG. Cependant, les performances s'en ressentiront.

LES PRINCIPES DE FORTHLOG

FORTHLOG tourne autour d'un principe essentiel, celui de $variable. Si un mot est rencontré et que ce n'est ni un nombre ni un mot du vocabulaire courant, il est transformé en $variable. Pour créer une $variable, il suffit de la nommer dans la base des faits. Pour exemple, soit un circuit TTL constitué de quatres portes NAND (type TI 7401):



Circuit:



On déclare les faits A1, A2 et AS en tapant dans le bloc destiné aux faits:

```
AFF Position de la borne A1 - 0 ou 1 - " A1 $AFN
AFF Position de la borne A2 - 0 ou 1 - " A2 $AFN
AS                     ;
```

Les règles contrôlant les sorties sont décrites à partir de la table de vérité d'une porte NAND:

A1	A2	AS
0	0	1
0	1	1
1	0	1
1	1	0

Ce qui donne en FORTHLOG dans les règles de contexte:

```
SI A1 0 $=  &  A2 0 $=
ALORS 1 AS $AF-N
SI A1 0 $=  &  A2 1 $=
ALORS 1 AS $AF-N
SI A1 1 $=  &  A2 0 $=
ALORS 1 AS $AF-N
SI A1 1 $=  &  A2 1 $=
ALORS 0 AS $AF-N
```

Et permet de declencher des règles résultat:

```
SI AS 0 $=
ALORS $AFF La borne AS est inactive
SI AS 1 $=
ALORS $AFF La borne AS est active
```

Si vous êtes patient, vous pouvez décrire un circuit beaucoup plus complexe, le dessiner et visualiser directement sur l'écran les différentes valeurs du circuit.

FORTHLOG traite les valeurs logiques, mais également les grandeurs numériques et les chaines de caractères. Aisni, en tenant compte de l'alimentation électrique dans l'exemple du circuit logique, on peut également définir le fait:

```
$AFF Valeur tension alim -en Volts x 10- " VOLTS $AFN
```

et exploité dans l'univers des résultats:

```
SI VOLTS 38 $<
ALORS AFF" Circuit sous-alimenté, tension < 3,8V " $STOP
```

Il faut entrer la tension en valeur entière. C'est pourquoi la valeur demandée est ajustée pour permettre un traitement efficace. Pour faire une action plus complexe au niveau de la regle déclenchée par une tension insuffisante, on peur un mot:

```
: TENSION_INSUFFISANTE
CR ." Votre tension d'alimentation est inférieure à 3,8 V"
CR CR ." Vérifiez les points suivants:"
CR ."    1 - intensité en sortie de régulateur"
CR ."    2 - valeur de la résistance tampon" CR ;
```

et utilisé dans la règle RESULTAT

```
SI VOLTS 38 $<
ALORS TENSION_INSUFFISANTE $STOP
```

Voilà, j'ai fait mes deux pages...

Les utilisateurs de FORTHLOG II sont invités à poser des questions, nous faire part des systèmes experts qu'ils mettent au point, exprimer des remarques et des suggestions. Selon les cas, nous les diffuserons dans JEDI ou en ferons part au(x) intéressé(s)(es). ■

HANDLER TU 58
par
Mme M.J. PEYRIN
RIOM LABORATOIRES - CERM

Voici un programme très professionnel et d'usage très spécifique réalisé par la Sté RIOM LABORATOIRES. Il permet de transférer des données sur un lecteur/enregistreur de cartouches TU 58 via l'interface série d'un PC1512.

0

```
\ interface TU58 de DEC avec PC1512  liaison RS232    16JUN87MJP
FORTH DEFINITIONS
HEX
VARIABLE COMPK E ALLOT          VARIABLE BUFBLC 200 ALLOT
VARIABLE NBYL    VARIABLE NBLL   VARIABLE PTRDP
VARIABLE SFLG    VARIABLE SNBDP  VARIABLE EPMES 0A ALLOT

CODE INIT 0 # AH MOV 0 # DX MOV E3 # AL MOV 14 INT NEXT C;

CODE WRCH 0 # DX MOV  AX POP 1 # AH MOV 14 INT NEXT C;
CODE RDCH 0 # DX MOV 2 # AH MOV 14 INT FF # AX AND 1PUSH NEXT C;

 : INTACTU INIT 4 WRCH 4 WRCH BEGIN RDCH 10 = UNTIL ;
 --)
```

6

```
\ initialisation du TU58                                16JUN87MJP
programme LECTURE OU ECRITURE d'un bloc TU soit 512 bytes
Certains tests ne sont pas mentionns dans ce programme
COMPK :buffer commande packet  BUFBLC:buffer de donnees
si nbre de donnees a lire ou a ecrire )512 redefinir BUFBLC
NBYL : nbre de bytes a lire ou ecrire NBLL:n°bloc
SFLG:flag operation SNBDP:tampon nbre de bytes PTRDP:pointeur
CODE INIT init liaison RS232 8bit 9600bauds pas de parite
1 bit de stop utilise interruption 20
CODE WRCH envoi 1 caractere DX:n°canal AL:carac AH:1
CODE RDCH lit 1 caractere DX:n°canal AH:2   AL:carac AH:etat

CODE INTACTU init TU58 envoi 2 fois 04 attend carac "continu"
```

1

```
\ calcul checksum ,init commande packet           15JUN87MJP
CODE CHSM (  snbdp sflg pt ---checksum)
 DI POP DX POP BX POP BL DH MOV BL BH MOV 1 # BH AND 1 # BH CMP
 0= IF BL DEC BH DEC 0 [BX+DI] AX MOV FF # AX AND CLC AX DX ADD
 0 # DX ADC THEN  BEGIN BX DEC BX DEC 0 # BX CMP 0)= WHILE
  0 [BX+DI] AX MOV CLC AX DX ADD 0 # DX ADC    REPEAT
 DX PUSH NEXT C;

: TCP ( fg---) COMPK 2 DUP SFLG C! OVER C! 0A DUP SNBDP C!
 OVER 1+ C! DUP 4 + 4 ERASE
 NBYL @ OVER 8 + ! NBLL @ OVER 0A + !
 DUP 2+ ROT OVER C! SNBDP C@ SFLG C@ ROT CHSM SWAP 0C + !
 0E 0 DO COMPK I + C@ WRCH LOOP ;
--)
```

7

```
\ calcul cheksum   init command packet                15JUN87MJP
CODE CHSM
addition des donnees 2 bytes par 2 bytes sur 16 bit
avec recuperation si retenue dans le LSB

: TCP init commande packet
transmission du commande packet
```

2

```
\ lecture-ecriture data packet,code succes          15JUN87MJP
: TRDP ( checksum---) SFLG C@ WRCH SNBDP C@ DUP WRCH 0 DO
 PTRDP @ I + C@ WRCH LOOP DUP FF AND WRCH FLIP FF AND WRCH ;

: INIL 2 TCP BUFBLC PTRDP ! ;
: RINIT  BEGIN 4 WRCH RDCH 10 = UNTIL ;

: DTNB NBYL DUP @ DUP 80 ) IF 80 DUP SNBDP C! - ELSE
  SNBDP C! 0 THEN SWAP ! ;

: STKDT ( ---snbdp checksum) RDCH DUP DUP SNBDP C! 0 DO
 RDCH PTRDP @ I + C! LOOP RDCH RDCH FLIP + ;

: PEPMES EPMES PTRDP ! STKDT 2DROP
  EPMES 1+ C@ DECIMAL . NBLL @ . HEX CR ;
--)
```

8

```
                                                       15JUN87MJP
: TRDP ecriture des donnees contenues dans le buffer et de la
 checksum

: INIL init command packet lecture et le buffer lecture
: RINIT reinitialisation TU58

: DTNB calcul du nbre de bytes a transmettre maxi 128
 et remise a jour du nbre restant

: STKDT lecture du nbre de bytes a lire et de ces donnees
 stockage de ces donnees dans le buffer et recuperation cheksum

: PEPMES lecture du END-PACKET
 et impression du code succes et du n° du bloc
```

3

```
\ Lecture d'un bloc ,positionnement                 15JUN87MJP
: LECTURE
 BUFBLC PTRDP !
 0 COMPK 3 + C! INIL
  BEGIN  RDCH DUP SFLG C! 2 = NOT WHILE SFLG C@ 1 =
   IF STKDT SWAP 1 PTRDP @ CHSM = NOT
    IF RINIT INIL ELSE SNBDP C@ PTRDP +! THEN
  THEN REPEAT PEPMES ;

: POSBLC 0 COMPK 3 + C! 5 TCP BEGIN RDCH 2 = UNTIL PEPMES ;
-->
```

9

```
\ Lecture  positionnement                            15JUN87MJP
: LECTURE pour plusieurs bloc redefinir BUFBLC
Necessite init NBLL,NBYL les donnees sont lues par paquet de 128
init commande packet lecture
lecture test si END-PACKET si DATA PACKET
 stockage des donnees ,calcul checksum si pas correcte
 reinitialisation  si OK init buffer suivant
continu lecture jusqu'a END-PACKET avec impression code succes

: POSBLC ( positionne TU58 au bloc specifie)
 necessite init de NBLL
```

4

```
\ ecriture TU58                                      15JUN87MJP
: SPECRIT
 1 COMPK 3 + C! 3 TCP
 BUFBLC PTRDP ! 0 SNBDP C! BEGIN  RDCH DUP 2 = NOT
 WHILE 10 = IF SNBDP C@ PTRDP +! 1 SFLG C! DTNB
 SNBDP C@ 1 PTRDP @ CHSM TRDP THEN REPEAT DROP PEPMES ;


: ECRITURE
 NBYL @ BEGIN  SPECRIT EPMES 1+ C@ 4 =
  WHILE  BUFBLC @ 2 NBYL ! INTACTU LECTURE
  BUFBLC ! DUP NBYL ! REPEAT DROP ;

DECIMAL
```

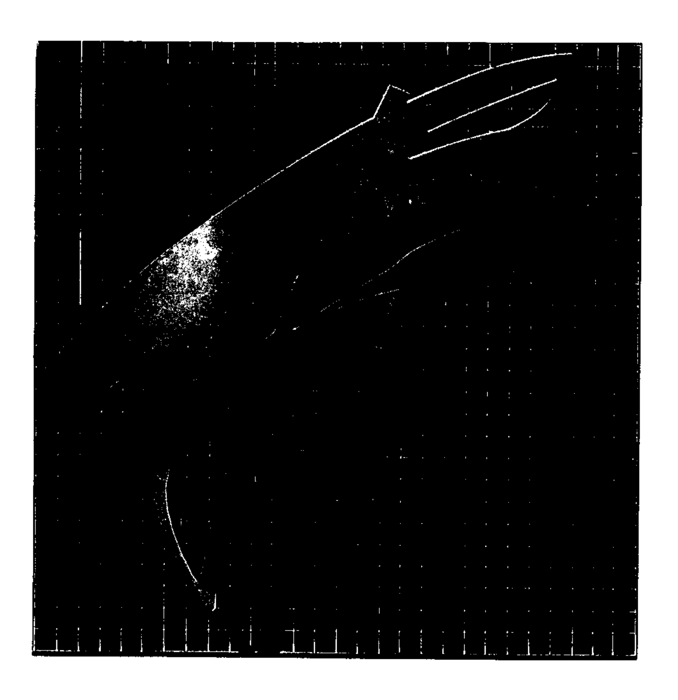
10

```
Ecriture                                             15JUN87MJP
: SPECRIT
 init commande packet ecriture
 init buffer ecriture test si pas END-PACKET
  alors si "continu" determine nbre de donnees a ecrire
  calcul checksum et transmet DATA-PACKET par paquet de 128 max
 jusqu'a END-PACKET et imprime code succes

: ECRITURE necessite init NBLL,NBYL
 les donnees sont ecrites par paquet de 128 bytes maxi
 si le code succes = not 0 pseudo lecture et reinitialisation
 pour ecriture
```

RL CERM

RIOM LABORATOIRES-CERM



F.A.U.S.T
Forum des Arts de l'Univers
Scientifique et Technique
Ville de Toulouse

19 - 24 octobre 1988
TOULOUSE PARC DES EXPOSITIONS

F83/MSDOS: Editeur Plein Ecran.

A. Jaccomard

0

Cet éditeur "plein écran" est une adaptation du VEDITOR contenu dans UNIFORTH, noyau Forth du domaine public. Il a été adapté à F83 et des fonctions ont été ajoutées.

Ces fonctions sont accessibles par les touches curseur, F1-F10 aussi par CTL. Elles sont auto-explicites, d'autant qu'un 'menu' se trouve affiché en permanence. Quelques explications ne seront peut-être pas inutiles :

- Sh-F1 permet l'accès à Forth, retour sous VEDIT en répondant 'N' à "On continue?", si les piles n'ont pas été modifiées; autrement, sortie; pour ré-entrer, tapez 'ED'.
- F2 déplace d'un coup un groupe de lignes.
- F4/Sh-F4 permettent le déplacement ligne par ligne entre blocs
- F3 suivi de 'nn' affiche le bloc 'nn'.
- F5/Sh-F5 sont les SPLIT et JOIN du F83 d'origine.
- F7/Sh-F7 et F8/Sh-F8 déplacent/effacent mot à mot.
- F9/Sh-F9 insère/supprime la ligne courante.

1

```
\ VEDIT, compatibilité: CASE à CASEND.                23Mar87JaD
 ONLY  FORTH  ALSO DEFINITIONS
 VOCABULARY VEDIT
: $CASE  R> DUP 2+ SWAP @ >R >R ;
: $=:     OVER =  IF  DROP  R> 2+ >R  ELSE  R> @ >R THEN ;
: $>:     OVER <  IF  DROP  R> 2+ >R  ELSE  R> @ >R  THEN ;
: $<:     OVER >  IF  DROP  R> 2+ >R  ELSE  R> @ >R  THEN ;
: $;;     R>  DROP ;
: NOCASE     DUP  ;
: CASE   COMPILE $CASE HERE 0 , ; IMMEDIATE
: =:     COMPILE $=:   HERE 0 , ; IMMEDIATE
: >:     COMPILE $>:   HERE 0 , ; IMMEDIATE
: <:     COMPILE $<:   HERE 0 , ; IMMEDIATE
: ;;     COMPILE $;;   HERE SWAP ! ; IMMEDIATE
: CASEND COMPILE R>  COMPILE 2DROP HERE SWAP ! ; IMMEDIATE
-->
```

2

```
\ VEDIT : PCGOTO, CLRWIN.                             23Mar87JaD
HEX
CODE PCGOTO  (S col lig -- )              \ positionne curseur.
    DX POP  DL DH MOV ( ligne ) BX POP  BL DL MOV ( colonne )
    2 # AH MOV  ( n° fonct. )  0 # BH MOV ( n° page écran )
    10 INT  NEXT C;
CODE CLRWIN  (S lig-col lig-col attr -- )  \ efface écran.
    600 # AX MOV ( AL=0, AH=06 )  BX POP ( BH=oct. attrib. )
    CX POP ( CH=lig. CL=col. coin sup. g. )
    DX POP ( DH=lig. DL=col. coin inf. d. )
    BP PUSH  10 INT  BP POP NEXT C;
CODE LIS-T  (S -- code-touche | 0 )
   8 # AH MOV  21 INT  AH AH XOR  1PUSH C;
DECIMAL
   2 22 +THRU
  24 27 +THRU   CR  BRILL .( VEDIT chargé.)  NORM  CR
```

3

- Ctl-W efface l'écran courant et est très efficace ...
- F10 propose 3 fonctions: F <txt> recherche la chaîne 'txt', R <txt1> <txt2> remplace 'txt1' par 'txt2', D <txt> détruit 'txt'; avec demande de confirmation. Le premier caractère entré est considéré comme délimiteur, ce n'est pas obligatoirement un espace.
- Sh-F10 fait une sauvegarde des blocs modifiés sans sortir.
- ESC sauve et sort, Ctl-X sort sans sauver.
- l'appui sur touche non affectée fait sortir de VEDIT.
- la bonne exécution des commandes vidéo suppose la présence sur la disquette des fichiers CONFIG.SYS et ANSI.SYS.

Ces blocs devraient être intégrés à UTILITY.BLK en remplacement de EDITOR, pour être compilés dans un nouveau noyau.

A défaut, VEDIT peut être chargé "par dessus" sans inconvénient autre que de place perdue.

A. Jaccomard, Mars 87.

4

```
\ VEDIT: variables.                                   23Mar87JaD
2VARIABLE  CLRSZ            VARIABLE  ATTR
 VARIABLE  #STLINES         VARIABLE  CURPOS
 VARIABLE  ?MODE            VARIABLE  DELIM
 VARIABLE  INSTGL           VARIABLE  CHANGED
: HOME    0 0 PCGOTO ;
: INIT    0  6223  CLRSZ  2! 0 ATTR ! ;
: CLPGE   CLRSZ  2@  ATTR @  CLRWIN ;
: VDO    27 EMIT 91 EMIT EMIT   109 EMIT ; \ ESC["n"m
: NORM   48 VDO ;
: BRILL  49 VDO ;
: DIM    52 VDO ;
: CLIGN  53 VDO ;
: INV    55 VDO ;

ONLY FORTH ALSO VEDIT DEFINITIONS
```

5

```
\ VEDIT: TABMEN.                                      15Mar87JaD

 CREATE TABMEN  6 75 * ALLOT  \ pr 6 lig. de 75 car.
 BLK @  BLOCK 5 64 * +  TABMEN  6 75 * CMOVE
\S
<-  -> CurUp  CurDn  Ins:bascIns  Ann:effCar  Tab  Home/^Home:d
ébLi/BlcFin/^Fin:finLi/Blc  DEL:effCarPré  F1:insEsp  F2:depLig
Blc  F3:sautBlocF4:transfLi  shF4:inslLi  F5/shF5:coupe/colle  F
6:LiPréc  shF6:effFinLigF7/shF7:Mpréc/eff  F8/shF8:Msuiv/eff  F9
/shF9:ajout/suppLig  shF1:cmdFtHF10:modeChaînes   shF10:Flush
     Sorties:  ESC:Sauve   ^X:Forth   Blocs:  PgUp/^PgUp:BlcPr
éc/1erBlc   PgDn/^PgDn:BlcSuiv/dernBlc   ^W:Wipe
```

6

```
\ VEDIT: déplacements curseur.                        16Mar87JaD

: P+     ROT + -ROT + SWAP ; (S n1 n2 n3 n4 -- n1+n3 n2+n4)
: GXY    CURPOS @ C/L /MOD ; (S -- col lig : pos. curs. )
: .CUR   GXY 4 1 P+ AT ;  (S -- ) \ place curs. ds fenêtre.
: !CUR   0 MAX B/BUF 1- MIN  CURPOS ! ; (S n -- )
: !.CUR  !CUR .CUR ;               (S n -- )
: +CUR   CURPOS @ + !CUR ; (S n -- ) \ déplace curs.
: L##    CURPOS @ C/L / ; (S -- lig ) \ lig. courante du curs.
: +LIN   L## + C/L * !CUR ; (S n -- ) \ ajoute 'n' lig.
: +.CUR  +CUR .CUR ; (S n -- ) \ dépl. curs.
: .HOM   0 !.CUR ;      \ positionne curs. en début fenêtre.
: .CURR  CURPOS @ 1+ DUP B/BUF <   \ curs. 1 pas à dr.
    IF  DUP  CURPOS !  .CUR  ELSE  DROP  THEN ;
: .CURL  CURPOS @ 1- DUP  0>=    \ curs. 1 pas à g.
    IF  DUP  CURPOS !  .CUR  ELSE  DROP  THEN ;
```

9

```
\ VEDIT: FWD, BKWD.                                   15Mar87JaD

: FWD    \ rech. mot suivant.
   SCR @ BLOCK  B/BUF CURPOS @   DO   I OVER +  C@  BL <>
       IF  B/BUF I    DO  I OVER + C@ BL =
             IF  I !.CUR  LEAVE  THEN
            LOOP   LEAVE
      THEN  LOOP  DROP ;

: BKWD   \ rech. mot précédent.
   SCR @ BLOCK CURPOS @  0  DO  CURPOS @ I -  2DUP + C@ BL <>
       IF  DUP 0   DO  2DUP I - + C@ BL =
             IF  I - !.CUR  LEAVE
        ;  THEN  LOOP  LEAVE
     THEN  DROP  LOOP  DROP ;
```

```
7
\ VEDIT: LTOS, MENU, cadre.

: LTOS  HERE 300 + #STLINES @ C/L * + ; \ "pile" de lig.
: MENU  TABMEN 6 0 DO
      0 I 18 + AT  I 72 * OVER + 72 TYPE  LOOP
    DROP  69 10 AT   ." Bloc# "
          65 17 AT FILE @ BRILL .FILE BRILL ;
: MENLL   NORM     \ ré-aff. dern. lig.
     0 23 AT  5 72 * TABMEN + 72 TYPE  BRILL ;

: SIDES  16 0 DO  0 I 1+ AT  I 2 .R BL EMIT
       179 EMIT  68 I 1+ AT  179 EMIT LOOP ;
: DTLINE  0 DO  196 EMIT  LOOP ;
: IDOTL 192 EMIT  64  DTLINE 217 EMIT ;
: SDOTL 169 EMIT  64  DTLINE 170 EMIT ;
: TBDOT   3 0 AT  SDOTL    3 17 AT  IDOTL ;
```

```
8
\ VEDIT: déplacements curseur.
: CLLINE 0 DO BL EMIT LOOP ;
: LINE  C/L B/BUF */MOD  SCR @ + BLOCK + ;
: TYLINE  4 OVER 1+ AT LINE C/L TYPE ;
: BLSTR  0 19 2DUP AT  72 CLLINE  AT ;
: BLCMD  0 23 2DUP AT  72 CLLINE  AT ;
: PRBLK      (S n -- )  \ aff. 'n' lig. ds fenêtre.
    DO  I TYLINE  LOOP ;
: .BLK       (S -- )     \ aff. blc courant ds fenêtre.
    BRILL  75 10 AT  SCR ?  16 0 PRBLK  .HOM ;
: TYCMD      (S -- )     \ aff. 1 lig. de la 'pile'.
    BLCMD  2 23 AT  #STLINES @ 2 .R
            7 23 AT  C/L TYPE .CUR ;
: VLST        \ init. en dessinant fenêtre.
    CLPGE  NORM  TBDOT  SIDES  MENU  #STLINES @
     IF  LTOS  TYCMD  THEN  .HOM ;
```

```
10
\ VEDIT: STAMP.                                                 23Mar87JaD
CREATE ID  11 ALLOT    ID 11 BLANK
: STAMP  ID  SCR @ BLOCK  C/L + 10  - 10  CMOVE ;
: ?STAMP  CHANGED @  IF  STAMP  THEN ;
: GET-ID  ID  11  -TRAILING  NIP  0=
     IF  CR  ." Entrez votre cachet: "
       11 0  DO  ASCII . EMIT  LOOP  11  BACKSPACES
       ID  11  EXPECT  THEN ;
: (QUI)    (S -- )   " JaD" ;  \ placez ici VOS initiales.
: QUI      (S -- )   (QUI) TYPE SPACE ;
: SET-ID H? IF (DATE) [ VEDIT ] ID SWAP CMOVE (QUI) ID 7 + SWAP
 CMOVE THEN   HELLO  ;
 ' SET-ID IS BOOT
: MODIFIED  ?STAMP  UPDATE ;
\ Cachet placé autom. en ligne 0 après BOOT et SAVE-SYSTEM.
\ Pour le supprimer : Sh-F1 puis CHANGED OFF.
```

```
11
\ VEDIT: ADDL, DELL, TGLINSERT.                                 23Mar87JaD

: ADDL   \ insère une ligne blanche.
    L## DUP 14  DO   I LINE DUP  C/L + C/L CMOVE  -1  +LOOP
    LINE  C/L  BLANK  MODIFIED  16 L## PRBLK ;

: DELL   \ supprime une ligne.
   L## 1+ 16 2DUP  <
 IF  SWAP  DO  I LINE DUP C/L - C/L CMOVE  LOOP
 ELSE  2DROP
 THEN  15 LINE C/L BLANK  MODIFIED  16 L## PRBLK  .CUR ;

: TGLINSERT  \ bascule d'insertion.
   0 24 AT  INSTGL @  1 XOR  DUP  INSTGL !
 IF  ." Insère"  ELSE  ."        "  THEN ;
```

```
12
\ VEDIT: PUTL, GETL, PSCR, DISPLAY.                             23Mar87JaD

: PUTL           \ place lig. courante ds 'pile'.
    LINE DUP  LTOS C/L CMOVE  1 #STLINES +!  TYCMD ;
: GETL           \ place en lig. cour. celle de la 'pile'.
   #STLINES @
  IF   LINE DUP  -1 #STLINES +!
 LTOS SWAP C/L CMOVE  4 L## 1+ AT C/L TYPE
  #STLINES @
    IF  LTOS C/L - TYCMD
   ELSE  MENLL   .CUR
     THEN  MODIFIED
 ELSE  DROP  THEN ;
: PSCR   71 10 AT  SCR @  .  2 SPACES  .HOM ; \ aff. n° blc
: DISPLAY  1 INSTGL !  TGLINSERT  0 CURPOS !  .BLK ;
```

```
13
\ VEDIT: TCLINE, !CHAR, CDEL, >CHAR, DELCHAR.                   23Mar87JaD
: TCLINE    (S f col lig -- )  \ aff. ds fenêtre.
   2DUP  4 1 P+ AT  LINE OVER + C/L ROT - -TRAILING  ROT
    IF  1+  THEN  TYPE  .CUR ;
: !CHAR     (S car -- )  \ place 'car' ds tampon et fenêtre.
   DUP  EMIT  SCR @  BLOCK  CURPOS @ +  C!  MODIFIED
    CURPOS @ 1+ DUP C/L MOD
 IF  CURPOS !  ELSE  DROP 1 +.CUR  THEN ;
: CDEL      (S -- )        \ efface car. ss curs et tampon.
   1 GXY  2DUP  LINE  DUP  63 + -ROT + 2DUP
     - OVER 1+ -ROT MOVE  BL SWAP  C!  TCLINE  MODIFIED ;
: >CHAR     (S car -- )  \ insère un car.
   GXY  2DUP  LINE OVER + SWAP  63 SWAP - OVER
   1+ SWAP MOVE  ROT !CHAR  0 -ROT  TCLINE   MODIFIED ;
: DELCHAR        \ efface car. précédent.
    GXY  DROP  IF  -1 +.CUR  CDEL  THEN   ;
```

```
15
\ VEDIT: chaînes.                                               23Mar87JaD

: !STRING    (S adr -- )  \ place une chaîne 'cptée' en adr.
  DUP   BEGIN
  >KEY  DUP  DELIM @ =  OVER  13 =  OR  \ DELIM ou CR.
    IF  DROP  1
   ELSE  DUP  8 =
       IF  DROP RUBOUT  1-
     ELSE  OVER  C! 1+
      THEN  0
    THEN
UNTIL   OVER - SWAP 1-  C! ;

 84 CONSTANT C/PAD
: TEXA  PAD  C/PAD + ;
: TEXB  TEXA C/PAD + ;
```

```
16
\ VEDIT: chaînes.                                               15Mar87JaD

: >FIND  >KEY   DELIM !  TEXA 1+ !STRING ;  \ entre ch. en TEXA
: >REP   >FIND  TEXB 1+ !STRING ;  \ entre ch. en TEXA et TEXB.
: MATCH    (S ad1 l1 ad2 l2 -- adr|0 )
  ROT OVER - DUP 0<
   IF  DROP 2DROP  0  ELSE  1+ 0
    DO 2 PICK 2 PICK 2 PICK  COMP
       IF  ROT 1+ -ROT  ELSE  2DROP  0  LEAVE  THEN
  LOOP
 IF  2DROP  0  THEN
  THEN ;
: FSTR  (S -- adr|0 ) \ rech. ds tampon chaîne de TEXA.
   CURPOS @  SCR @  BLOCK  OVER + B/BUF ROT -
   TEXA COUNT MATCH ;
```

14

```
\ VEDIT: CLREOL, DELSTR, ADDSTR, DELWD, >KEY, RUBOUT. 15Mar87JaD

: CLREOL       \ efface fin lig.
   CURPOS @ C/L  GXY  DROP - 0   DO   BL !CHAR  LOOP  !.CUR ;
: DELSTR   0 DO  CDEL  LOOP ;  (S n -- ) \ eff. 'n' car.
: ADDSTR   0 DO  DUP I + C@ >CHAR  LOOP  DROP ;
: ?OUI     (S -- f : -27=ESC, -1=O, 0=autre )
   KEY  DUP  27 =   IF NEGATE   ELSE
  SP@ 1 UPPER  ASCII O =  THEN ;
: REPSTR   0 DO  DUP I + C@ !CHAR  LOOP  DROP ; \ remplace ch.
: DELWD    \ efface un mot.
   SCR @ BLOCK  CURPOS @ +
  BEGIN   CDEL  DUP C@ BL = UNTIL  DROP ;
: >KEY   KEY DUP EMIT ;  (S -- k )
: RUBOUT  BL EMIT BS EMIT ;
```

18

```
\ VEDIT: chaînes.                                   15Mar87JaD
: MOVSTR   (S -- ) \ remplace ou efface chaîne.
    TEXB C@  TEXA C@  -
CASE   0 =:  TEXB COUNT  REPSTR ;;
  DUP 0 >:  TEXB 1+  TEXA C@ 2DUP  REPSTR  +  SWAP  ADDSTR ;;
        ABS  DELSTR  TEXB COUNT  REPSTR
CASEND ;
: FIXSTR  (S -- 0 -1|-1 -1) \ traite chaînes selon réponse.
  ?OUI DUP -27 <>  IF
   0 SWAP  0=   IF  1 +CUR   ELSE   ?MODE @
CASE  -1 =:  TRUE ;;
       0 =:  TEXA C@  DELSTR ;;
       1 =:  MOVSTR ;;
CASEND  THEN
ELSE  DROP  BLCMD  .HOM  TRUE TRUE  THEN ;
```

19

```
\ VEDIT: PROMPT.                                    17Mar87JaD
: PROMPT    \ mode Chaînes.
BEGIN
    FSTR  ( -- adr|0 ) DUP  0=
  IF  DROP INV BLCMD  ." Pas trouvé...bloc suivant?"  NORM
       ?OUI  IF  BLCMD 1 SCR +!  DISPLAY  0
           ELSE  TRUE TRUE                  \ sortie et .HOM.
            THEN
  ELSE  SCR @ BLOCK - !CUR  BLCMD
        INV  ." Cette chaîne? (O/N/ESC): "  NORM
         :CUR  FIXSTR
  THEN
UNTIL
  NORM  MENU  BRILL   IF  .HOM  ELSE  .CUR  THEN ;
```

20

```
\ VEDIT: ?IN, INPUT, $EXECUTE, COMFTH.              23Mar87JaD
: ?IN          (S -- n )  \ empile le nbr 'n' entré au clavier.
    TEXA 1+  !STRING  BL  TEXA COUNT +  C!
    TEXA 0 DUP ROT  CONVERT  2DROP ;
: INPUT        (S -- N )  \ comme ?IN, sans CR ni TEXA.
    QUERY 32 WORD  NUMBER?  IF  DROP
        ELSE  2DROP RECURSE  THEN ;
: $EXECUTE  (S adr lgr -- )   \ exécute les cdes de la chaîne.
   DUP  #TIB !  TIB SWAP MOVE  BLK OFF  >IN OFF  INTERPRET ;
: COMFTH    (S -- )   \ par sh-F1 : exécute une cde Forth.
    0 18 2DUP AT 6 80 * CLLINE  AT
    R@  !CSP  QUERY  TIB  SPAN @  $EXECUTE
    CR ." OK"  CR  ." On continue ? "  ?OUI
     IF  DROP  RECURSIVE  COMFTH
    ELSE  ?CSP  R@ <>  ABORT" Pile Retour modifiée"
    VLST  BRILL  DISPLAY  THEN ;
```

17

```
\ VEDIT: chaînes.                                   23Mar87JaD

: STRINFO
    INV  20 18 DO 0 I AT  72 CLLINE  LOOP
    10 18 AT
  ." Mode Chaîne:  F, D, R [sep]CH1[sep]CH2[sep] " ;

: STRMODE           \ réponse -1, 0, 1 dans ?MODE.
    STRINFO  BLSTR  >KEY  SP@ 1 UPPER
CASE  ASCII F =:  >FIND  -1 ;;
      ASCII D =:  >FIND   0 ;;
      ASCII R =:  >REP    1 ;;
      NOCASE  =: ."  Entrée incorrecte!!" 7 EMIT  650 MS
                     RECURSIVE STRMODE ;;
CASEND  ?MODE ! ;
```

21

```
\ VEDIT: SCR#.OK?, JBLK.                            23Mar87JaD
: SCR#.OK?   (S n -- : vérif. n° écr. valide )  DUP 0<
   IF   DROP  0   ELSE   DUP  CAPACITY - 1+ DUP 0>
    IF  BLCMD  BRILL  ." Extension min. de "  DUP .
       ."  nécessaire - entrez quant. désirée : "
       ?IN  BLCMD  2DUP  >
     IF  BLCMD  7 EMIT  ." INSUFFISANT"  2DROP
        750 MS  BLCMD .HOM
    ELSE   MORE  DROP  SCR !
    THEN
  ELSE  DROP  SCR !
 THEN  1     \ pour boucle 'UNTIL'.
THEN ;
: JBLK     \ saut vers l'écr. spécifié.
  75 10 AT  ?IN  SCR#.OK?
  IF  DISPLAY  ELSE  PSCR  THEN ;
```

22

```
\ VEDIT: SPLIT, JOIN.                               15Mar87JaD

: SPLIT        \ "coupe" lig. au curs, la place sur lig. suiv.
   GXY  LINE  DUP  C/L +   ( col adli adlisuiv )
    -ROT  +  2DUP  -       ( adlisuiv adcar lgr )
    3DUP  >R SWAP R>       ( adcar adlisuiv lgr )
   1 +LIN ADDL  CMOVE     \ de adcar à adligsuiv sur lgr.
 BLANK DROP  -1 +LIN  16 L## PRBLK  .CUR  ;

: JOIN         \ "colle" lig. suiv. à lig. courante.
    GXY  LINE DUP  C/1 + -ROT
     +  2DUP  - 3DUP  CMOVE  NIP BLANK
    16 L## PRBLK .CUR ;
```

23

```
\ VEDIT: TRANSLIN, BRING.                           23Mar87JaD

: TRANSLIN    (S scr l1 l2 -- )  \ transf. lig. sur lig. cour.
     SCR @ >R   ROT SCR !  2DUP  >  IF  SWAP  THEN
  2DUP  ?DO I LINE LTOS C/L CMOVE 1 #STLINES +! -1 +LOOP
    R> SCR !
   SWAP - 1+ 0  ?DO  L##  GETL  1 +LIN  LOOP MODIFIED ;
: BRING      \ commande transfert de lignes.
       BLCMD INV  ." du bloc ? "          INPUT
                  ."  1ère ligne ? "      INPUT
                  ."  dernière ligne ? "  INPUT
     ."   D'accord ? (O/N/ESC)"    ?OUI   BRILL
CASE  -27 =:  MENLL    BRILL .HOM ;;
       -1 =:  TRANSLIN MENLL BRILL .CUR  ;;
        0 =:  2DROP R> 2DROP  RECURSIVE BRING ;;
CASEND ;
```

```
24
0 \ VEDIT: ?MOVE.                                      23Mar87JaD
: ?MOVE        \ cdes déplacmts curs., lig. et blocs.
CASE      75   =:  -1  +.CUR  0    ;;   ( <- )
          77   =:   1  +.CUR  0    ;;   ( -> )
          80   =:  C/L +.CUR  0    ;;   ( cursB )
          72   =: -64 +.CUR  0     ;;   ( cursH )
          73   =: SCR @  1- SCR#.OK?
                 IF  DISPLAY  ELSE  PSCR  THEN  0  ;; ( PgUp )
          81   =: SCR @  1+ SCR#.OK?
                 IF  DISPLAY  ELSE  PSCR  THEN  0  ;; ( PgDn )
          61   =:  JBLK    0        ;;    ( F3 )
          64   =: -1 +LIN .CUR  0  ;;    ( F6 )
          63   =:   SPLIT   0       ;;    ( F5 )
          88   =:   JOIN    0       ;;    ( sh-F5 )
          79   =:  1 +LIN  -1 CURPOS +! .CUR  0 ;; ( Fin )
          71   =:  L#  C/L *  !.CUR    0       ;; ( Home ) -->
```

```
25
0 \ VEDIT: suite ?MOVE, ?CNTRL.                        23Mar87JaD

       93   =:    SAVE-BUFFERS  0  ;;    ( sh-F10 )
DUP  CASEND ;

: ?CNTRL
CASE
      23  =:  SCR @ BLOCK  B/BUF  BLANK .BLK  MODIFIED ;; ( ^W )
      27  =:  FLUSH DARK          NORM DARK QUIT ;; ( ESC )
      24  =:  DARK  EMPTY-BUFFERS  NORM DARK QUIT ;; ( ctl-X )
       8  =:  DELCHAR                            ;; ( DEL )
      13  =:  1 +LIN  .CUR                       ;; ( Enter )
       9  =:  10 +CUR  .CUR                      ;; ( --> )
 DUP  CASEND ;
```

```
26
0 \ VEDIT: CINS.                                       23Mar87JaD

: CINS                        \ insertion chaîne de car.
   TGLINSERT  .CUR
BEGIN    LIS-T
CASE
    0    =:  LIS-T  82  =       ( Ins )
             IF  TGLINSERT  .CUR  1  ELSE  0  THEN  ;;
    8    =:  DELCHAR  0  ;;     ( DEL )
       DUP  31 >   IF  >CHAR  ELSE  DROP  THEN  0  DUP
CASEND
 UNTIL   MODIFIED ;
```

```
27
\ VEDIT: ?CHAR.                                        23Mar87JaD
: ?CHAR                                 \ cdes d'édition de texte.
CASE
    '66  =:   FWD  0  ;;     ( F8 )      \ mot suivant.
     65  =:  BKWD  0  ;;     ( F7 )      \ mot précédent.
     82  =:  CINS  0  ;;     ( Ins )     \ mode "Insertion".
     83  =:  CDEL  0  ;;     ( Annul )   \ efface sous curs.
     90  =:  BKWD  DELWD  0  ;;   ( sh-F7 ) \ eff. mot préc.
     91  =:        DELWD  0  ;;   ( sh-F8 ) \ eff. mot suivant.
     59  =: BL   >CHAR    0  ;;   ( F1 )    \ insère un blanc.
     84  =:  COMFTH    0  ;;      ( sh-F1 ) \ cde Forth.
     68  =: STRMODE  PROMPT  0  ;;  ( F10 ) \ mode "Chaînes"
 DUP CASEND ;
```

```
28
\ VEDIT: ?LINE.                                        23Mar87JaD
: ?LINE
CASE
     67  =:  ADDL   .CUR  0  ;; ( F9 )          \ ajoute 1 lig.
     92  =:  DELL   .CUR  0  ;; ( sh-F9 )       \ eff. lig.
     62  =:  L#    PUTL  .CUR  0  ;; ( F4 )    \ transf. lig.
     87  =:  L#    GETL  .CUR  0  ;; ( sh-F4)  \ ins. lig.
     89  =:  CLREOL  0  ;; ( sh-F6 )           \ efface fin lig
     64  =: -1 +LIN .CUR  0  ;; ( F6 )    \ curs. lig. préc.
     60  =:  BRING  0  ;;       ( F2 )    \ transf. blc/blc.
    132  =:  0 SCR ! .BLK 0 ;;  ( ^PgUp )        \ 1er bloc.
    118  =: CAPACITY 1- SCR ! .BLK 0 ;; ( ^PgD) \ dern. bloc.
    117  =: B/BUF 64 - !.CUR 0 ;; ( ^Fin)  \ curs. dern. lig.
    119  =:    .HOM     0       ;; ( ^Home) \ curs. 1ère lig.
 DUP  CASEND ;
```

```
29
\ VEDIT: Entrée et sortie: (VEDIT), EDIT, ED.          23Mar87JaD
VARIABLE  LDERR   LDERR  OFF
: (VEDIT)    (S n -- )  \ édite écran 'n'.
    INIT  DUP  SCR !  0 #STLINES !  VLST  SCR#.OK?
  IF  DISPLAY  LDERR @  IF  !.CUR  BKWD  LDERR OFF
                         ELSE  .HOM  THEN
 ELSE  DARK  ABORT  THEN
BEGIN  LIS-T  ?DUP
  IF  DUP  32 <
     IF  ?CNTRL  ELSE  !CHAR  MODIFIED  THEN  0
  ELSE  LIS-T  ?MOVE  ?LINE   ?CHAR
  THEN  UNTIL ;    ONLY  FORTH ALSO DEFINITIONS
: ED  [ VEDIT ]  GET-ID  SCR @ (VEDIT) ;
: EDIT  1 ?ENOUGH  SCR !  ED ;
: (?OU) DISK-ERROR @  0=  IF [ VEDIT ] LDERR ON  ED THEN ;
  ' (?OU) IS WHERE
```

LOGIQUE ET SYSTEMES EXPERTS
par Marc PETREMANN

LA LOGIQUE

Dès l'antiquité, les philosophes grecs discutaient de la logique et de son comportement sur le discours. Ils étaient passés maîtres dans l'art de triturer les doubles négations, triples négations et phrases conditionnelles. Exemple:

- je dit que je suis grec
- je ne dit pas que je ne suis pas grec

Ou pire encore, des propos récursifs:

- il est vrai que je suis un menteur

Or, la limite de tout discours sur la logique, et surtout dans ces temps lointains, tenait dans le manque de formalisme du traitement de la logique, ce qui faisait dire à certains:

- un cheval est mortel
- l'homme est mortel
- donc, l'homme est un cheval

Les philosophes grecs oubliaient un peu que certaines structures logiques aboutissant à une même conclusion ne permettaient quand même pas d'affirmer que les tenants de deux affirmations avaient des propriétés identiques. Voici un cas arithmétique permettant de conclure à l'égalité de deux opérations:

- 5+5 donne 10
- 12-2 donne 10
- donc, 5+5 équivaut à 12-2

Mais ce résultat n'est valide que pour des valeurs spécifiques. Il peut en être de même si on essaie certaines opérations en portant notre attention sur l'opérateur:

- 2+2 donne 4
- 2*2 donne 4
- donc, l'addition équivaut à la multiplication.

Tout l'art du mathématicien consiste donc à dégager les règles intervenant dans les propriétés des opérations. Or, pour exprimer ces propriétés, il faut faire appel à un langage situé à un niveau supérieur aux symboles analysés.

Ce n'est que au siècle dernier que BOOLE[1] formalisa sous une forme mathématique les règles de la logique. Ses travaux et ceux de DE MORGAN ont abouti à définir les théorèmes permettant de définir la validité d'un théorème. Pour simplifier, la logique permet de savoir si un théorème a une solution avant de chercher cette solution.

Comment formalise-t-on la logique? Prenons dans le désordre, une affirmation et une condition:

soit P la condition
soit Q l'affirmation

Si la validité de l'affirmation Q ne dépend que de la seule condition P, on peut écrire:

si P est vrai, alors Q est vrai
si P n'est pas vrai, alors Q n'est pas vrai

Dans ces énoncés, notre écriture n'est pas encore assez formalisée. Empruntons une écriture plus mathématique:

P = Q
nP = nQ

Le signe = ne prenant ici un sens d'implication que dans le seul contexte de la logique, le signe n signifiant la non vérité ou le contraire. Exemples de contraires:

nP contraire de P
0 contraire de 1

0 et 1 sont contraires dans un système symbolique ou deux symboles seulement existent. On pourrait remplacer 0 et 1 par n'importe quels autres symboles:

- contraire de +
! contraire de ?
A contraire de B, etc...

En automatique[2], la négation est marquée par un trait au-dessus de la condition:

$\bar{a}$

Dans la vie courante, on peut déjà appliquer ceci, pour exemple, à la fonction remplie par l'interrupteur commandant l'allumage d'une ampoule électrique.

soit L la lampe, a l'interrupteur, on développe le système logique

$L = \bar{a}$ la lampe est éteinte
$L = a$ la lampe est allumée

Si la lampe sert à signaler un évènement se produisant en divers points d'un mécanisme où des capteurs adéquats sont installés, on pourra écrire:

L = a
L = b
L = c etc...

Donc, si L dépend de a ou b ou c, on écrira en automatique:

L = a + b + c

si chaque variable prend un état, 0 ou 1, il suffit qu'une seule des variables passe à 1 pour que L passe à 1:

L = 1 + 1 + 0

signifie que L = 1, le signe + n'ayant pas le sens que lui prête l'arithmétique, l'addition, mais un sens logique exprimé par la conjonction OU (ou OR en anglais).

Si maintenant notre lampe dépend de deux conditions et que les deux conditions doivent être remplies simultanément, on écrira en automatique:

L = a * b

Le hasard fait bien les choses, car ici, l'application arithmétique ou logique du symbole donne les mêmes résultats:

aritha.	logique
0x0=0	0*0=0
0x1=0	0*1=0
1x1=1	1*1=1

Ce qui permet, en automatique de simplifier l'écriture:

L = ab le * signifiant ici la conjonction logique ET (AND en anglais) et devenant implicite.

Ainsi, pour une condition plus complexe:

L = a * b + a * c

et s'écrivant aussi sous la forme:

L = ab + ac

On peut procéder à la mise en facteur:

L = a(b+c)

forme d'écriture parfaitement admise en automatique et que l'on traduirait sous une forme moins symbolique par:

Si a est vrai et que b est vrai ou c est vrai

Alors l est vrai

ou en FORTHlog II:

SI A & B OU C
ALORS L

Et le OU EXCLUSIF me diriez-vous! Eh bien il n'existe pas. Le OU EXCLUSIF (XOR en anglais abrégé) n'est que l'image d'une structure logique composée à partir des fonctions ET OU et NON. C'est le cas du montage électrique correspondant au va-et-vient. L'allumage de la lampe est vérifié pour la formule suivante:

$$L = a \bar{b} + \bar{a} b$$

Ce qui s'exprime en bon français par "L est vrai si a est vrai ou b est vrai mais pas quand les deux sont vrais.

Toute la logique utilisée dans le mécanisme des ordinateurs est dérivée de ces trois opérations logiques. Toute fonction complexe peut être exprimée à partir d'une fonction plus simple. Ce qui diverge ensuite sont les notions de logique combinatoire et logique séquentielle.

LA LOGIQUE COMBINATOIRE

En logique combinatoire, tout système est vérifié par une combinaison de une ou plusieurs variables logiques. Cette forme de logique est très utilisée dans l'étude des ensembles, théorie mise au point par CANTOR*. Les patatoïdes informes se croisant ne sont qu'une autre formulation de la logique combinatoire.

La logique combinatoire ne tient pas compte des états transitoires, ce qui peut poser quelque problème quand on veut vérifier un système du type suivant:

as = bs
bs = as

qui est le système résultant de l'assemblage de deux portes NAND montées tête bêches:

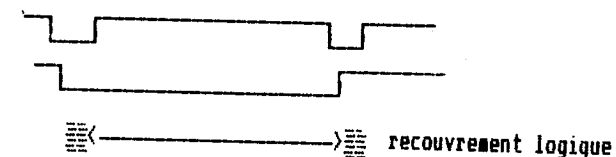
Un tel système est dit oscillant. En mathématique on dira qu'il est récursif.

Un autre cas de perturbation issu des transitions sont les états pouvant générer un résultat aléatoire. Toujours dans le domaine électrique, lors de la manoeuvre d'un interrupteur, la lame de l'interrupteur passe par un état transitoire qui n'est ni vrai ni faux. Cet état non prévu dans la logique mathématique doit être neutralisé en modifiant l'ordre des séquences d'essais d'un circuit logique. C'est le code binaire AIKEN.

ordre binaire normal	AIKEN	séquence:
000	000	1
001	001	2
010	011	3
011	010	4
100	110	5
101	111	6
110	101	7
111	100	8

Dans le code binaire normal, la composition d'un nombre obéit à l'opération logique b3*b2*b1 (b3 et b2 et b1). En passant d'une séquence à la suivante, dans certains cas plusieurs variables changent d'état en même temps. Dans le code binaire AIKEN, cet inconvénient disparait, ce qui supprime les états transitoires indésirables et générateurs d'aléas technologiques.

En logique électronique, on fait également appel à des diagrammes logiques (chronogrammes) où des temporisations sont générées pour tenir compte de ces états transitoires indésirables. Exemple:



Si les deux diagrammes avaient le front descendant simultanément en début de signal, il pourrait se produire dans certaines situations des cas où le second signal serait descendant avant le premier, situation indésirable résolue en temporisant le déclenchement du signal. Les erreurs logiques matérielles dépendent des tolérances et propriétés physiques des systèmes exploitant la logique (rebonds dans les commutateurs, capacités parasites sur composants, induction et self-induction sur lignes, etc...).

Pour résoudre les cas complexes et extraire la formule logique contrôlant un système, on utilise en automatique divers outils comme les matrices des états logiques et les tableaux de KARNAUGH.

La limite permettant une exploitation sans faille de ces outils est rapidement atteinte par le nombre de variables à traiter rendant les tableaux de KARNAUGH inexploitables quand un grand nombre de variables interviennent. Le nombre de combinaisons possibles est exprimé par la formule $N=2^v$ où v est le nombre de variables.

LOGIQUE SEQUENTIELLE

On peut exploiter les états transitoires pour "mémoriser" cet état et le conserver dans une variable. Prenons pour exemple un bouton poussoir. J'appuie dessus et ma lampe s'allume; je relâche le bouton poussoir et la lampe reste allumée. Pour que la lampe reste allumée, il faut que le changement d'état du bouton soit pris en compte par un organe de mémorisation. Cet organe, un relais à verrouillage en l'occurence, est lui même une variable pour la lampe. Ainsi détermine-t-on la suite logique:

X = a
X = x
L = x

En faisant référence à lui-même, on peut simplifier l'écriture de X en déclarant:

X = a + x
l = x

Si on met un autre bouton b en série sur le circuit x, on coupe X en déclarant:

$$X = a + x \bar{b}$$

ce qui remet X au repos quand on appuie sur b.

LA LOGIQUE NEGATIVE

Dans certaines situations, le nombre de cas où le résultat des combinaisons aboutissant à un résultat faux devient restreint, il est plus intéressant de tenir compte de ces cas.

Dans la vie courante, la logique négative peut être un facteur de sécurité. Le système de freinage WESTINGHOUSE est un cas de logique négative: s'il y a de la pression, les freins sont débloqués, s'il n'y a pas de pression, les freins se bloquent. Ainsi, en cas de rupture dans la continuité d'un attelage ferroviaire, le tuyau d'alimentation maintenant la pression dans les freins étant rompu, le freinage s'amorce et le train s'arrête.

En électronique, les circuits logiques, notamment en technologie TTL, exploitent beaucoup la logique négative.

Ces circuits se retrouvent dans toutes les parties matérielles des ordinateurs (composants de la série 74xx, ou 54xx pour le matériel aux normes militaires).

Dans un système formel, moins il y a d'exceptions, moins le nombre de règles à appliquer est élevé. S'il ne reste qu'une exception, on traitera en priorité l'exception en déclarant

```
Si <condition>
Alors le système est faux
si le système est NON faux
alors il est vrai.
```

La deuxième règle servant à valider toutes les autres situations.

Certaines formes logiques comme NON(A+B) sont développées sous la formes NON(A) ET NON(B), soit:

$\overline{A+B}$ $\quad$ $\overline{A}.\overline{B}$

C'est l'application de la loi de DE MORGAN*.

LOGIQUE ET SYSTEMES EXPERTS

La logique dans un système expert applique toutes les règles de la logique à des énoncés du type **SI condition ALORS action**. Selon le type de système expert utilisé, les conditions peuvent être des expressions exploitant des grandeurs arithmétiques, des chaînes de caractères, des valeurs booléennes ou des **prédicats.**

Concrètement, tout système informatique traite dans un langage quelconque des propositions logiques à partir desquelles diverses actions sont envisagées. Une proposition du type

SI j'ai acheté mon billet de train
ALORS je peux voyager
SI je suis à l'heure à la gare
ALORS je peux prendre le train
SI j'ai acheté mon billet de train ET je suis à l'heure à la gare
ALORS je peux monter dans le train

est nettement plus explicite qu'une proposition du type:

```
BILLET? IF VOYAGER ON  THEN
RETARD? IF PRENDRE ON  THEN
VOYAGER @ PRENDRE @ AND IF MONTER ON  THEN
```

Mais les mécanismes logiques restent similaires et sont indpendants de la nature du problème à traiter. Si je remplace train par allumette et billet par cigarette, on peut parfaitement résoudre les propositions suivantes:

SI j'ai acheté ma cigarette d'allumette
ALORS je peux voyager
SI je suis à l'heure à la gare
ALORS je peux prendre l'allumette
SI j'ai acheté ma cigarette d'allumette ET je suis à l'heure à la gare
ALORS je peux monter dans l'allumette

ce qui est un peu confus pour notre entendement mais est tout à fait logique (encore un peu de confiture avec votre dinosaure?... et LEWIS CARROL, prend-il du sucre dans son écrou?...)

Voyons comment exposer des propostions logiques et les résultats. Pour ce faire, nous mettrons en abscisse les variables, en ordonnée les résulats des règles dont dépendent les variables:

```
variables    a b
système    L ■ ■
```

Le signe ■ indique que L dépend de l'état des variables a et b

Si on introduit plusieurs variables, on obtient le tableau suivant:

```
variables    a b c
système    L ■ ■
           M   ■ ■
```

Le résultat d'une condition peut devenir lui-même condition d'une autre règle:

```
variables    a b c l
système    L ■ ■
           M   ■ ■
           P   ■   ■
```

la variable l résultant de l'état du système L. Donc P dépend de l qui dépend de L dépendant de a et b

Enoncées dans cet ordre, l'état de P peut être analysé par n'importe quel programme informatique. Mais si on modifie l'ordre des règles, un seul passage ne sera pas suffisant pour déterminer l'état de tout le système. Enonçons le contenu du dernier diagramme:

```
SI a ET b
ALORS L
SI b ET c
ALORS M
SI b ET l
ALORS P
```

On ne tiendra pas compte dans cette formulation des différences de caractères majuscules-minuscules. On déclare comme vraies les propositions a, b et c dans une base de faits connus et vérifiés:

on teste la première règle SI a ET b, le résultat l est vrai,
on teste la seconde règle SI b et c, le résultat m est vrai,
on teste la troisième règle, SI b et l, l étant déclaré vrai par la première règle, le résultat p est vrai.

Modifions l'ordre des règles:

```
SI b ET l
ALORS P
SI a ET b
ALORS L
SI b ET c
ALORS M
```

on teste la première règle SI b ET l, le résultat p ne peut être trouvé car l n'a pas encore été déclaré par aucune règle précédente.
on teste les règles suivantes, le résultat est inchangé par rapport aux précédentes conclusions.

En modifiant l'ordre des règles, on a modifié le résultat. Or cette différence n'existe que si on fait un seul passage dans la liste des règles. Un second passage donne une solution à la première règle.

Le nombre de passages à effectuer est contrôlé par la **saturation** de toutes les règles, c'est à dire lorsque toutes les règles ont été vérifiées, que le résultat soit déclaré vrai ou faux. C'est le principe de fonctionnement du langage PROLOG.

En partant de faits connus, nous vérifions une série de règles jusqu'à saturation. Le déroulement des opérations est effectué en descendant. C'est le **chainage avant.** Mais comment connaître les conditions contrôlant un résultat.

Partons du résultat P:

P dépend de b et l
b ne dépend que de lui-même
l dépend de a et b.

```
P --> b l
        L --> a b
```

La recherche des conditions dont dépend le résultat d'une

règle, c'est à dire une hypothèse, s'appelle le **chaînage arrière**.

Le chaînage arrière n'est possible qu'en logique des propositions (calculs des prédicats). Si on fait intervenir des grandeurs numériques dans les parties conditions, seul le chaînage avant reste cohérent.

Le chaînage avant délivre des conclusions multiples en fonction d'un certain nombre de données.

Le chaînage arrière délivre les propriétés permettant de vérifier une hypothèse donnée.

LA RECURSIVITE

La **récursivité** n'est pas un **mécanisme obligatoire** pour construire et faire fonctionner un système expert. Il existe des systèmes experts écrits en BASIC qui donnent des résultats honorables[5]. Les premiers langages informatiques permettant de traiter des listes de manière récursive et disponibles sur des micro-ordinateurs sont PASCAL, LISP et LOGO. PROLOG est le dernier né de la série et semble avoir la faveur des développeurs.

L'ennui est que ni PASCAL, LOGO ou LISP ne permettent de générer des systèmes experts ouverts, c'est à dire offrant la possibilité de travailler en logique et d'exécuter des instructions de bas niveau en intégrant les deux possibilités de manière intime.

LES SYSTEMES EXPERTS

Un système expert est avant tout une série de règles à partir desquelles un programme de traitement appelé **moteur d'inférence** va émettre des conclusions. Ces règles sont données en vrac et le moteur d'inférence va vérifier les règles les unes après les autres.

Le principe d'action d'un moteur d'inférence est simple à priori, mais c'est la méthode de représentation de la connaissance qui définit les performances d'un système expert.

Si le système expert se limite au seul calcul des prédicats, on aura des règles de la forme:

```
SI CIGARETTE
ALORS A-DE-QUOI-FUMER
SI ALLUMETTE OU BRIQUET
ALORS A-DU-FEU
SI A-DE-QUOI-FUMER ET A-DU-FEU
ALORS PEUT-FUMER
```

Si l'analyse porte sur le contenu de la chaine alphanumérique située après SI, ALORS et les conjonctions OU et ET, on peut devenir plus explicite:

```
SI j'ai une cigarette
ALORS j'ai de quoi fumer
SI j'ai des allumettes
OU j'ai un briquet
ALORS j'ai du feu
SI j'ai de quoi fumer
ET j'ai du feu
ALORS je peux m'en griller une
```

Mais attention, une coquille, une lettre en trop ou en moins, un accent oublié et les chaines ne sont plus semblables, ce qui perturbe le résultat. Pour éviter ce genre de problème, il faudra gérer des algorithmes traitant la pré-équivalence de chaines. Exemple:

soit deux chaines alpanumériques désignées par CH1 et CH2
- réduire tous les espaces multiples à un espace
- transformer tous les caractères accentués en leur équivalent non accentués
- transformer ou ne pas tenir compte des differences de caractères MAJ/min
- admettre au moins une erreur (ou plus...) par mot ou groupes de mots (TELLE =p= TELE) selon la taille de la chaine
- rechercher dans une librairie tous les diminutifs et abréviations syntaxiques (ST =p= SAINT, / =p= SUR, STE =p= SOCIETE, etc..)

Ces options pouvant être prises en compte ou non selon le degré de pré-équivalence recherché.

Un système expert admettant un haut niveau de pré-équivalence admettra des similitudes entre ces deux énoncés:

```
SI je vais à ILLKIRCH-GRAFFENSTADEN
ALORS je ferais un détour par NIDERSCHAEFFOLSHEIM
SI je vais a ILKIRCH GRAFFENSTADEN
ALORS ....
```

L'annuaire électronique (disponible par le 11) accepte la notion de pré-équivalence en découpant en syllabes et en effectuant une comparaison phonétique:

```
BOULANGERIE =p= BOULENJERI
        BOU =p= BOU
        LAN =p= LEN
        GE  =p= JE
        RIE =p= RI
```

Si vous envisagez de créer un système expert capables de telles performances, vous avez du pain (=p= pin) sur la planche.

Rapidement, on peut envisager de créer un système expert capable d'effectuer des traitements et des tests sur des grandeurs numériques et alphanumériques:

```
SI TEMPERATURE > 45
ALORS JE RISQUE UNE INSOLATION CARABINEE
```

Le prédicat TEMPERATURE doit accepter plusieurs types. Un type logique, un type numérique. Si votre système expert est écrit en PASCAL, vous aurez déjà du fil à retordre. En LOGO ou en LISP ça ira encore:

```
DONNE "TEST 45
EC :TEST            affiche 45
DONNE "TEST [ CHIEN CHAT POULE ]
EC :TEST            affiche CHIEN CHAT POULE
```

On oublie BASIC. En FORTH, il faut définir un nouveau type de variable. C'est ce que fait FORTHLOG en créant des $VARIABLE et en **instanciant** leur contenu. La simple déclaration d'une $VARIABLE dans les faits l'instancie comme prédicat ayant une valeur "vraie".

```
faits FORTHLOG:     RHUM COCA VODKA GIN MARTINI

règles FORTHLOG:

SI RHUM COCA
ALORS CUBA-LIBRE
SI MARTINI CITRON
ALORS MARTINI-ON-THE-ROCKS
```

Votre bar ne contenant pas de citron, vous ne pourriez vous faire un MARTINI-ON-THE-ROCKS.

Ici, la règle ne donne pas les quantité mais seulement un PREDICAT. Si vous avez du RHUM et du COCA (ce qui a été déclaré dans les faits, donc vrai), la partie action définit un nouveau prédicat CUBA-LIBRE.

FORTHLOG II permet de mélanger les prédicats et les définitions FORTH selon un autre principe très simple:

```
si un mot n'existe pas
  on vérifie si c'est un nombre
     et on l'empile
  sinon on crée une $VARIABLE
  sinon
     on exécute le mot
  finsi.
```

Une $VARIABLE déjà créée est exécutée de la **même manière** qu'un mot FORTH. La partie action d'une **règle résultat** dans

FORTHLOG sera de la forme:

```
SI CUBA-LIBRE
ALORS RECETTE-CUBA-LIBRE
```

en ayant pris soin de définir le mot RECETTE-CUBA-LIBRE:

```
: RECETTE-CUBA-LIBRE  ( --- )
 DARK  ." RECETTE DU CUBA LIBRE" CR CR CR
 ." 1/4 RHUM"  CR
 ." 3/4 COCA COLA" CR ;
```

Dans FORTHLOG II.8°, un nouveau type de données est défini, le type **DOCUMENT**. Ce type de données permet de définir un mot FORTH auquel est rattaché un fichier ASCII de taille quelconque:

```
DOCUMENT CUBA-LIBRE CUBALIBR.DOC
```

Et le fichier CUBALIBR.DOC contient la recette complète du cocktail:

" Pour faite un CUBA-LIBRE comme dans les îles:
- prendre un grand verre
- y mettre deux gros glaçons
- jeter dedans une peau de citron vert
- verser un doigt de rhum blanc
- remplir avec du coca cola
Appréciez, consommez avec modération."

Un fichier **<fich>.DOC** peut avoir une taille quelconque. Il est créé à l'aide d'un éditeur de texte. Son affichage sur l'écran vidéo est réalisé par blocs de 24 lignes de 80 caractères.

En FORTHLOG II, on peut exécuter un mot ou une séquence de mots FORTH n'importe où dans la base de connaissances:

- dans les faits
- dans les règles univers, contexte et résultat.

Les faits ne sont validés qu'une seule fois. Les règles univers et contexte sont vérifiées jusqu'à saturation. Les règles résultat ne sont vérifiées qu'une seule fois.

D'autres générateurs de systèmes experts permettent d'exécuter des instructions de manière algorithmique. C'est le cas de GURU, mais les instructions sont spécifiques à GURU et ne permettent pas un interfaçage au système de manière aussi profonde que ce que réalise FORTHLOG.

■

REFERENCES

Manuel PROLOG pour THOMSON

MATHEMATIQUES POUR INFORMATICIENS,
série SCHAUM ed MAC GRAW HILL

INTELLIGENCE ARTIFICIELLE ET SYSTEMES EXPERTS
Adrien LESCORT, ed CEDIC NATHAN

SUPPORT DE COURS GURU

HISTOIRE DES MATHEMATIQUES
Encyclopédie LAROUSSE

1. George BOOLE (GB, 1815-1864). A publié en 1847 son premier ouvrage, THE MATHEMATICAL ANALYSYS OF LOGIC, ouvrage contemporain de celui publié par DE MORGAN, FORMAL LOGIC OR THE CALCULUS OF INFERENCE. Ses travaux sont à la base des techniques exploitant la logique (automatisme, informatique) mais aussi les mathématiques.

2. La notation utilisée en logique est dépendante du domaine dans laquelle elle est exploitée. Pour des raisons de facilité typographique, nous nous limiterons à celle utilisée en automatisme, notre propos n'étant pas de faire un cours de mathématique appliquée.

3. Georg CANTOR (D, 1845-1918). A publié en 1869 une thèse sur les transformations des formes ternaires quadratiques. A entretenu avec Richard DEDEKIND une abondante correspondance où toutes les bases de la théorie des ensembles ont été exprimées, ce qui a donné lieu à la parution d'une série d'articles dans ce domaine dès 1873 et entre 1895 à 1897.

4. Augustus DE MORGAN (GB, 1806-1871). Avec G.BOOLE est un des fondateurs de la logique mathématique. Auteur de FORMAL LOGIC (1847).

5. J.KRUTCH "Expériences d'Intelligence Artificielle en BASIC" Ed EYROLLES

6. FORTHLOG II.8 en cours de préparation. La disquette de mise à niveau sera proposée à tous ceux ayant déjà acquis la version FORTHLOG II.

DE NOUVEAUX UTILITAIRES
POUR LES MANIPULATIONS SUR LA PILE
ET LE PASSAGE DE PARAMETRES EN FORTH

par
Paul BARTHOLDI
Observatoire de Genève (CH)
(1982 Rochester Forth Conference)

Depuis l'année dernière à GRENOBLE, nous avons considérablement accru les sécurités dans l'utilisation de FORTH en rajoutant des trappes pour la détection des erreurs de compilation. Mais nous n'avons rien fait pour faciliter les manipulations de pile et beaucoup d'erreurs en proviennent. La seule "nouvelle" commande est ROLL de KPNO (et ROCK de St Addrews). Elles sont très utiles mais pas plus aisées que les précédentes.

Le schéma proposé plus loin est vraiment différent de toutes les opérations de manipulation de pile. Il facilite l'utilisation et la lecture ultérieure. En fin de compte, il est conçu, après adaptation à FORTH, de manière similaire à ce qui est disponible sur les compilateurs ALGOL ou PASCAL. (voir le concept DISPLAY expiqué par Organick dans "Computer System Organisation, série B 5700-6700", Prentice Hall, 1973).

UTILISATION:

Dans une procédure, en principe avant d'y accéder mais pas nécessairement, déclarez le nombre <n> de paramètres utilisés par cette procédure avec la commande

<n> PARAMETER

Puis dans la procédure, et n'importe où après la déclaration, vous pouvez empiler une **valeur** quelconque d'un de ces paramètres anec la commande PARn

où n est un nombre compris entre 1 et 17, o une lettre comprise entre A et P.

PAR1 (ou PARA) se réfère au paramètre le plus bas dans la pile, PARN au sommet de la pile après exécution de la commande PARAMETER, et ceci indépendemment de tout élément rajouté depuis sur le sommet de la pile de données. Ceci restera vrai tant que nous ne rencontrerons pas le mot ";" qui ferme la procédure.

Ce schéma peut être suivi aveuglément par une procédure s'appelant elle-même et utilisant ce schéma. Il peut être utilisé indéfiniment par une procédure récursive (voir exemple).

EXEMPLE:

Pour exemple, nous allons coder en FORTH la procédure donnée par DIJKSTRA pour résoudre le problème des tours de HANOI (dans "Structured Programming", Academic Press, 1972). DIJKSTRA donne le code suivant en notation ALGOL:

```
begin procedure movetower (integer value m,A,B,C)
       begin if m=1 then movedisk (A,C)
                    else begin movetower (M-1,A,C,B);
                               movedisk (A,C);
                               movetower (M-1,B,A,C);
                                   movetower (N,1,2,3)
```

La procédure est récursive, avec appel de quatres paramètres par valeur. La traduction en FORTH donne:

```
: HANOI  4 PARAMETER
         PAR1 1 = IF
         PAR PAR4 MOVEDISK
              ELSE
         PAR1 1- PAR2 PAR4 PAR3 ITSELF
                  PAR2 PAR4 MOVEDISK
                  PAR1 1- PAR3 PAR2 PAR4 ITSELF
              THEN
         4 NDROP ;

: MOVEDISK  SWAP "  DE " . " VERS " . CR ;

: HANOI CR CR "  TOURS DE HANOI AVEC " DUP . " DISQUES"
         CR CR "   IL FAUT DEPLACER LES DISQUES : " CR CR
         1 2 3 HANOI  ;
```

```
4 HANOI

  TOURS DE HANOI AVEC 4 DISQUES
   IL FAUT DEPLACER LES DISQUES :

     DE 1  VERS 2
     DE 1  VERS 3
     DE 2  VERS 3
     DE 1  VERS 2
     DE 3  VERS 1
     DE 3  VERS 2
     DE 1  VERS 2
     DE 1  VERS 3
     DE 2  VERS 3
     DE 2  VERS 1
     DE 3  VERS 1
     DE 2  VERS 3
     DE 1  VERS 2
     DE 1  VERS 3
     DE 2  VERS 3
```

L'utilisation des seuls mots NOVER (équivalent de PICK) ou S' ou toute construction similaire rend le programme beaucoup plus complexe et plus difficile à mettre au point et à comprendre:

```
: MOVEDISK  SWAP " DE " . " VERS " . CR ;

: HANOI RECUR 4 NOVER ! = IF 3 NOVER 2 NOVER MOVEDISK
ELSE 4 NOVER 1 - 4 NOVER 3 NOVER 5 NOVER ITSELF
3 NOVER 2 NOVER MOVEDISK
4 NOVER 1 - 3 NOVER 5 NOVER 4 NOVER ITSELF THEN
DROP DROP DROP DROP ;

: TOURD'HANOI CR CR CR " DEPLACE LE DISQUE " CR CR 1 2 3
HANOI CR CR  ;
```

(Pour ceux qui considèrent que la récursivité est un luxe, écrivez ce programme produisant le même résultat sans passer par la récursivité!)

Remarques:

- ITSELF est un appel récursif à la procédure courante. Il est défini par : ITSELF LAST @ , ; IMMEDIATE

- PAR1 équivaut à m ,
  PAR2 à A, PAR3 à B et PAR4 à C

- PARAMETER redéfinit un nouvel environnement à chaque exécution de HANOI, directement ou par ITSELF.

- portez votre attention sur le fait que chaque exécution de PARAMETER utilise trois mots sur la pile de retour (plus l'adresse de retour de la procédure en cours). La taille de la pile de retour doit être prévue en conséquence.

IMPLEMENTATION:

L'implémentation utilise une simple zone de stockage temporaire dont la trace est sauvegardée sur la pile de retour et pointe sur un "paquet de données". RP ne peut être utilisé si sa valeur est changée par DO ou une commande similaire.

Le paquet de paramètres consiste en trois mots:

- une adresse de retour de pseudo terminaison de procédure laquelle détruit le paquet, puis exécute le retour normal de la procdure.

- un pointeur sur la première donnée de la pile (PAR1).

- une copie du précédent contenu de TEMP qui sera réinstallé en fin de procédure.

L'installation a été réalisée sur un UNIVAC 1108 et un HP 21MX FORTH. Mais les commentaires vous aiderons à réaliser une adaptation sur votre système personnel:

```
0 INT 2              zone de stockage pointant sur le
                     dernier paquet de données entré.
```

(code pour détruire le paquet de données de la pile de retour)

```
ORC1 RP I) Load      copie la valeur dans la pile de retour
           STR2      stockage temporaire, réinstallation du
                     précédent pointeur.
           POP       dépile deux motsde la pile de retour
           NEXT      sort de la procédure
```

(pseudo-terminaison)

```
HERE  INT3           copie la prochaine adresse dans INT3
                     (voir PARAMETER)

AD01                 adopte le précédent code
                     met une fin à cette pseudo-terminaison.
                     Nous entrons ici en fin de procédure
                     principale.
```

CODE PARAMETER

```
SP    Load           Ajoute le nombre de paramètres au
                     pointeur de pile
S)    Add            (nous prélevons l'adresse du premier
                     paramètre)
RP    Push           On l'empile sur la pile de retour.
      RCL2           Chargement de la valeur dans la zone de
                     stockage temporaire.

RP    Push           On l'empile sur la pile de retour.
      RCL3           Chargement de l'adresse de la pseudo-
                     terminaison
RP    Push           On l'empile sur la pile de retour.
      Pop            dépile le nombre de paramètres de la
                     pile.
```

(code pour empiler au sommet de la pile le nième paramètre)

```
ORC0         RCL2 Charge l'adresse d'un paquet.
      2      Add  Charge l'adresse du pointeur du 1er
                  paramètre.
      I)     Load Prend l'adresse du premier paramètre.
      IC I)  Add  Ajoute le contenu du mot suivant PARn,
                  lequel est (n-1).
                  Nous avons maintenant l'adresse du nime
                  paramètre.
      I)     Load Prend la valeur du paramètre
      IC     Incr Incrémente IC pour passer au-dessus de
                  (n-1).
             Push Empile la valeur au sommet de la pile
```

```
: PARN  HERE 2 + @
        400 / 17 AND
        1- MINUS
        LIT, AD00 , ;  IMMEDIATE
```

PARN doit être exécuté en compilation.

Implémentation complète de PARAMETER et PARN sur HP 21MX

```
BLOCK 21 (25B)
( PARAMETER ET PARN OU N = 1 .. 16 OU A .. P  P.BDI )
                  HERE 0 ,
ORC1 RP I) LDA, DUP STA, RP LDA, TWO ADA, RP STA, NEXT ,
HERE  AD01 ' ; ,  HERE SWAP , SWAP
CODE PARAMETER SP LDA, S) ADA,
               RP LDA, -ONE ADB, B I) STA,
               DUP LDA, -ONE ADB, B I) STA,
               -ONE ADB, DUP STB,
               SWAP LDA, B I) STA, RP STB, POP ,
ORC0  LDA, INA, INA, A I) LDA,
   IC I) ADA, A I) LDA, IC ISZ, PUSH ,
: PARN HERE 2 + @ 400 / 17 AND 1- MINUS LIT, AD00 , ; ,
IMP PARN
;S
;S
;S
```